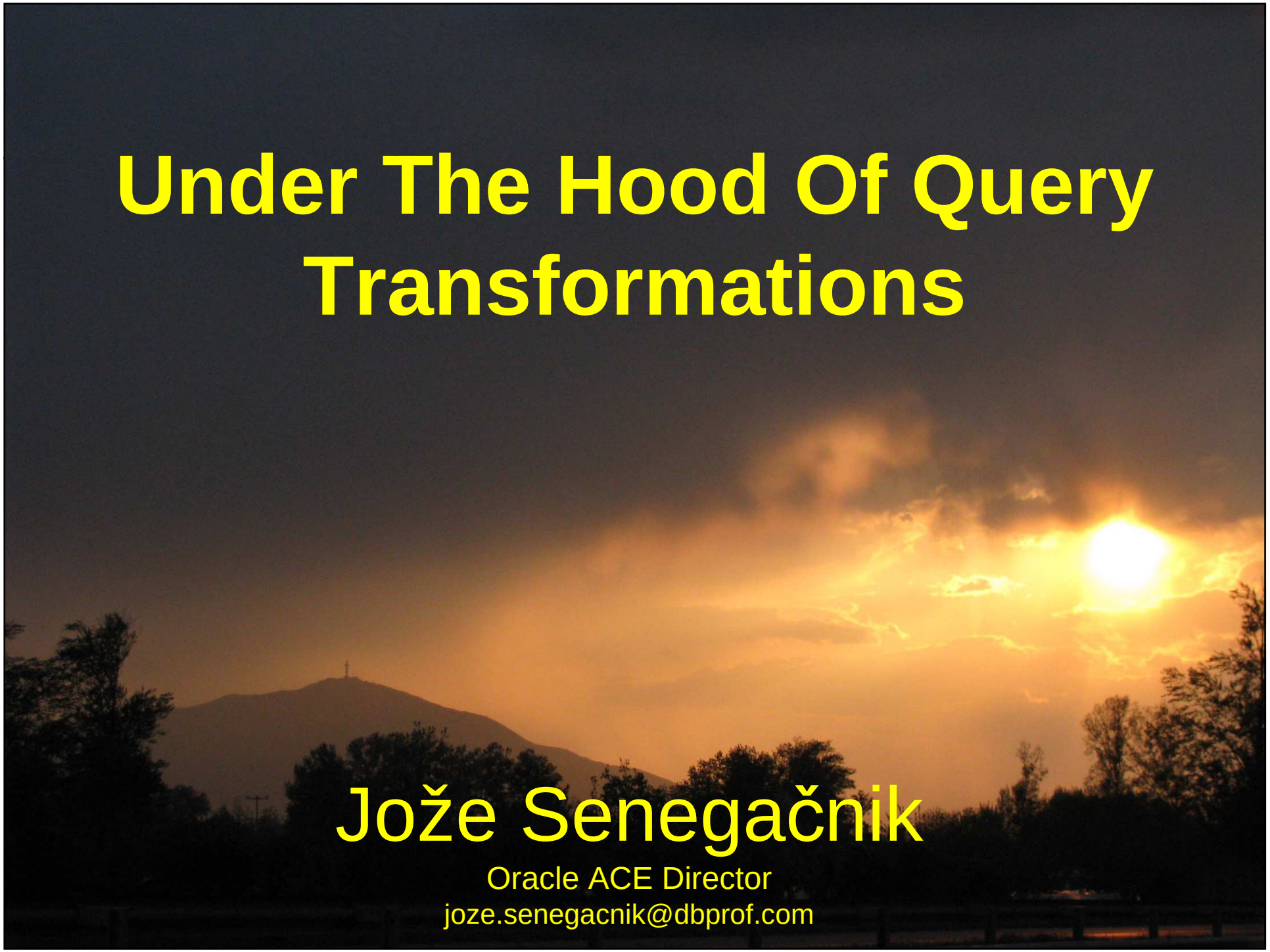


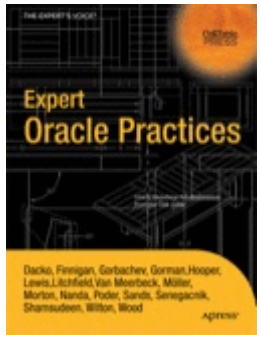
Under The Hood Of Query Transformations

Jože Senegačnik
Oracle ACE Director
joze.senegacnik@dbprof.com

About the Speaker

Jože Senegačnik

- Registered private researcher
- First experience with Oracle Version 4 in 1988
- 25 years of experience with Oracle RDBMS.
- Proud member of the OakTable Network www.oaktable.net
- Oracle ACE Director
- Co-author of the OakTable book “Expert Oracle Practices” by Apress (Jan 2010)
- VP of Slovenian OUG (SIOUG) board
- CISA – Certified IS auditor
- Blog about Oracle: http://joze-senegacnik.blogspot.com

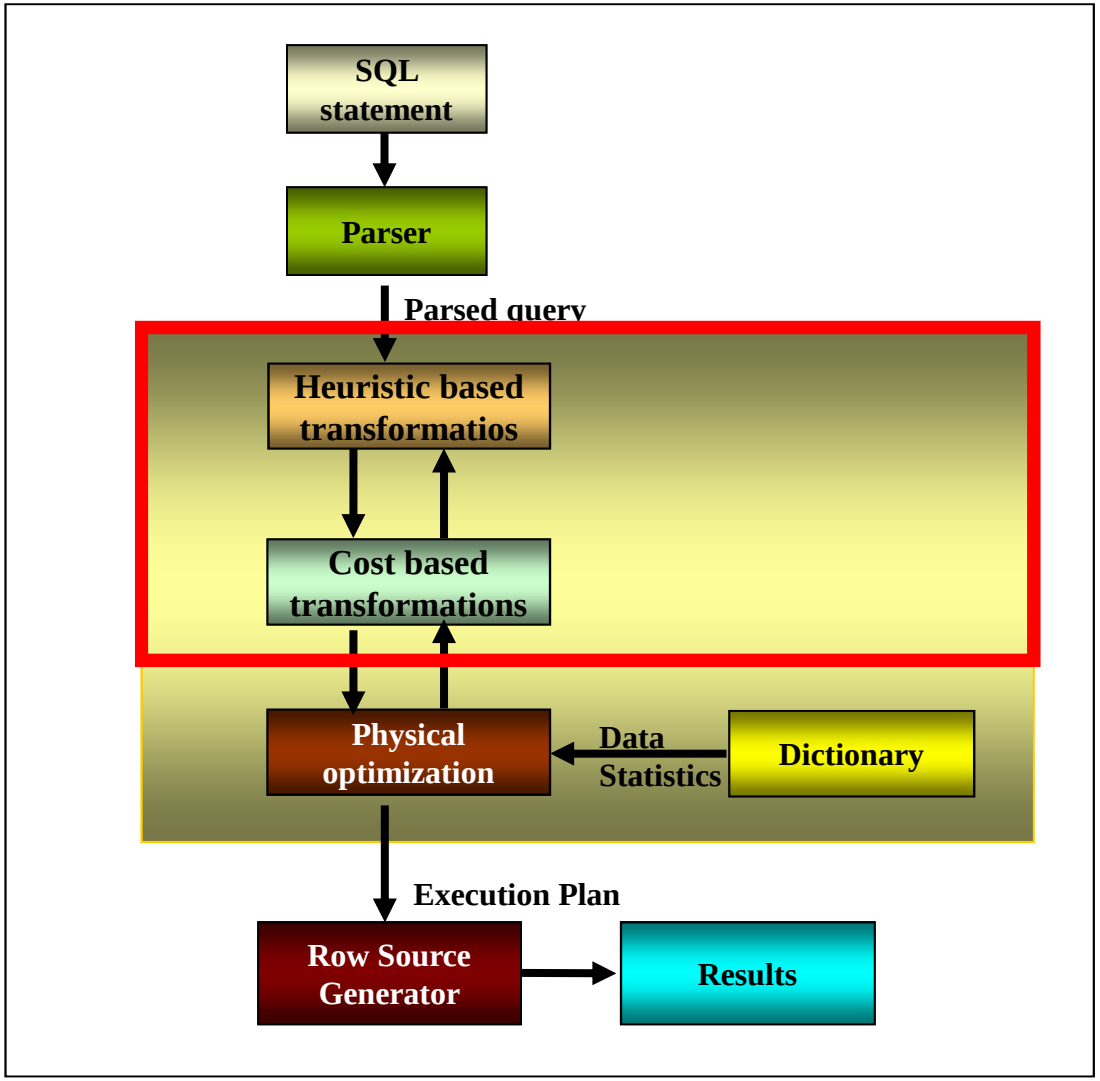


- PPL(A) – private pilot license PPL(A) / instrument rated IR/SE
- Blog about flying: http://jsenegacnik.blogspot.com
- Blog about Building Ovens, Baking and Cooking: http://senegacnik.blogspot.com

Agenda

- We will discuss the following transformations (as named in CBO trace):
  - Common Sub-Expression Elimination
  - JPPD - join predicate push-down
  - FPD - filter push-down
  - PM - predicate move-around
  - CVM - complex view merging
  - SPJ - select-project-join
  - SJC - set join conversion
  - SU - subquery unnesting
  - OBYE - order by elimination
  - CNT - count(col) to count(*) transformation
  - JE - Join Elimination
  - JF - join factorization
  - SLP - select list pruning
  - DP - distinct placement
- Oracle 12c (12.1)
  - Scalar Subquery Unnesting transformation
  - Null Accepting Semi-Joins

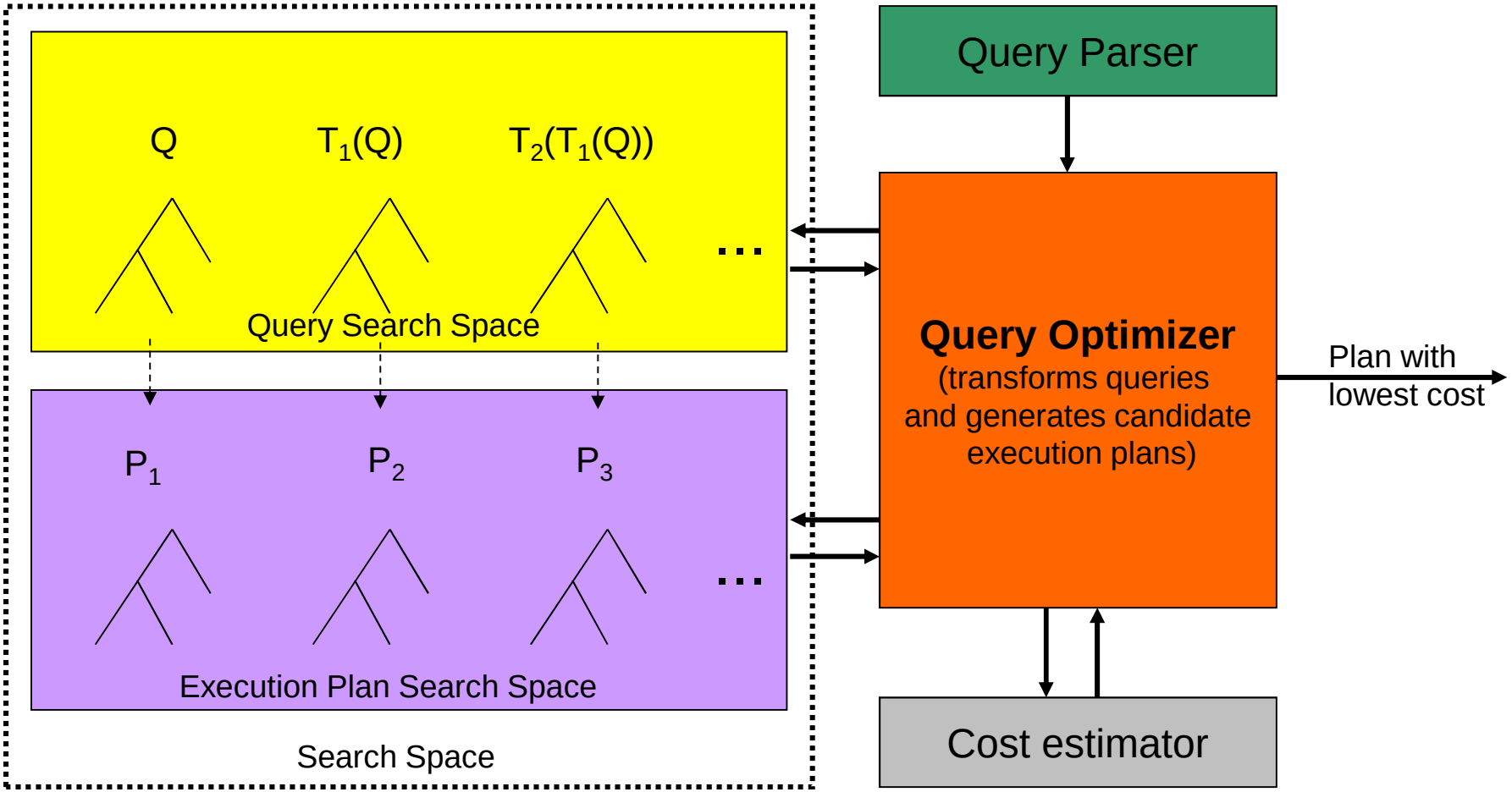
SQL Statement Processing



Query Optimization

- Query optimization is performed in two phases
  1. **Logical optimization** (query transformation)
  2. **Physical optimization** – finds information
     - Possible access method to every table (full scan, index lookup,...)
     - Possible join method for every join (HJ, SM, NL)
     - Join order for the query tables (join(join(A,B), C)

Query Optimization



Why Query Transformations?

- The goal of transformation is to enhance the query performance.
- Transformation generates semantically equivalent form of statement, which produces the same results, but significantly differs in performance.
- Transformation rely on algebraic properties that are not always expressible in SQL, e.g, anti-join and semi-join.

Transformations

- CBO supports different approaches:
  - Automatic – which always produce a faster plan
  - Heuristic-based
    - Prior to 10gR1
    - Assumption – produce faster plan most of the time
    - User has to set parameters or use hints
  - Cost-based
    - Since 10gR1
    - Transformation does not always produce a faster query
    - Query optimizer costs non transformed query and transformed query and picks the cheapest form
    - No need to set parameters or use hints
- Transformation may span more than one query block

Query Transformations

Common Sub-Expression Elimination

- The purpose is to remove duplicate predicates and avoid processing the same operation several times.
- This is a heuristic-based query transformation.

```
select * from sh.customers
where    (cust_city_id=52297 and CUST_GENDER='M')
      or (cust_city_id=52297);
```

```
select * from sh.customers where cust_city_id=52297;
```

SU - Subquery Unnesting

SU - subquery unnesting

- Original may be sub-optimal because of multiple, redundant re-evaluation of the sub-query
- Un-nesting
  - sub-query converted into an inline view connected using a join, then merged into the outer query
  - Enables new access paths, join orders, join methods (anti-/semi-join)
- A wide variety of un-nesting
  - Any (IN), All (NOT IN), [NOT] EXISTS, correlated, uncorrelated, aggregated, group by
- Some are automatic; what used to be heuristic-based is cost-based since Oracle10g
- Related optimizer hints: UNNEST, NO_UNNEST

SU - unnesting NOT EXISTS

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM customers c
WHERE NOT EXISTS
   (SELECT 1
    FROM orders o
    WHERE o.cust_id = c.cust_id);
```

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM customers c, orders o
WHERE c.cust_id A= o.cust_id;
```

Execution Plan for NOT EXISTS

```
SQL> select cust_id,cust_first_name,cust_last_name
       from customers c
       where not exists ( select 1 from sales s where s.cust_id = c.cust_id);
```

```
-----------------------------------------------------------
| Id  | Operation                          | Name           |
-----------------------------------------------------------
|   0 | SELECT STATEMENT                   |                |
|*  1 |  HASH JOIN ANTI                    |                |
|   2 |   TABLE ACCESS FULL                | CUSTOMERS      |
|   3 |   PARTITION RANGE ALL              |                |
|   4 |    BITMAP CONVERSION TO ROWIDS     |                |
|   5 |     BITMAP INDEX FAST FULL SCAN    | SALES_CUST_BIX |
-----------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("S"."CUST_ID"="C"."CUST_ID")
```

Tables are from SH demo schema.

SU - unnesting EXISTS

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM customers c
WHERE EXISTS
    (SELECT 1
     FROM orders o
     WHERE o.cust_id = c.cust_id);
```

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name
FROM customers c, orders o
WHERE c.cust_id S= o.cust_id;
```

Execution Plan for EXISTS

```
SQL> select cust_id,cust_first_name,cust_last_name
       from customers c
       where exists ( select 1 from sales s where s.cust_id = c.cust_id);
```

```
-----------------------------------------------------------
| Id  | Operation                          | Name           |
-----------------------------------------------------------
|   0 | SELECT STATEMENT                   |                |
|*  1 |  HASH JOIN SEMI                    |                |
|   2 |   TABLE ACCESS FULL                | CUSTOMERS      |
|   3 |   PARTITION RANGE ALL              |                |
|   4 |    BITMAP CONVERSION TO ROWIDS     |                |
|   5 |     BITMAP INDEX FAST FULL SCAN    | SALES_CUST_BIX |
-----------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("S"."CUST_ID"="C"."CUST_ID")
```

Tables are from SH demo schema.

SU - unnesting aggregated sub-query

```
SELECT distinct p.prod_id, p.prod_name
FROM products p, sales s
WHERE p.prod_id = s.prod_id
AND s.quantity_sold < (SELECT AVG (quantity_sold)
                       FROM sales
                       WHERE prod_id = p.prod_id);
```

```
SELECT distinct p.prod_id, p.prod_name
FROM products p, sales s,
     (SELECT AVG (quantity_sold) as avgqnt, prod_id
      FROM sales
      GROUP BY prod_id) v
WHERE p.prod_id = s.prod_id
AND s.quantity_sold < v.avgqnt
AND v.prod_id = s.prod_id;
```

What CBO Really Does is ...

```
SELECT DISTINCT P.PROD_ID ITEM_1,
       P.PROD_NAME ITEM_2,
       CASE WHEN S.QUANTITY_SOLD <
            AVG(S.QUANTITY_SOLD) OVER ( PARTITION BY S.PROD_ID)
            THEN S.ROWID END VW_COL_3
FROM   SH.SALES S,SH.PRODUCTS P
WHERE  P.PROD_ID=S.PROD_ID
```

```
-------------------------------------------
| Id  | Operation                | Name     |
-------------------------------------------
|   0 | SELECT STATEMENT         |          |
|   1 |  HASH UNIQUE             |          |
|*  2 |   VIEW                   | VW_WIF_1 |
|   3 |    WINDOW SORT           |          |
|*  4 |     HASH JOIN            |          |
|   5 |      TABLE ACCESS FULL   | PRODUCTS |
|   6 |      PARTITION RANGE ALL |          |
|   7 |       TABLE ACCESS FULL  | SALES    |
-------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("VW_COL_3" IS NOT NULL)
   4 - access("P"."PROD_ID"="S"."PROD_ID")
```

FPD – Filter Push Down

FPD – Filter Push Down (1)

```
select distinct c4
from

   (select /*+ no_merge */ c4, count(*) cnt
    from t1 group by c4) a

where a.cnt > 100
```

```
-------------------------------------+-----------------------------------+
| Id  | Operation             | Name | Rows  | Bytes | Cost  | Time      |
-------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT      |      |       |       |     4 |           |
| 1   |  VIEW                 |      |     1 |    13 |     4 |  00:00:01 |
|*2   |   FILTER              |      |       |       |       |           |
| 3   |    HASH GROUP BY      |      |     1 |     3 |     4 |  00:00:01 |
| 4   |     TABLE ACCESS FULL | T1   |  1000 |  3000 |     3 |  00:00:01 |
-------------------------------------+-----------------------------------+
Predicate Information:
----------------------
2 - filter(COUNT(*)>100)
```

- a.cnt > 100 is pushed inside subquery

View Merging

View Merging

- Views are created for several reasons
  - Security
  - Abstraction (factorize same work performed by many queries)
  - Describe business logic
- However, they are used in different contexts
  - Filter on a view column
  - Join to tables or other views
  - Order by or group by on view column(s)
- View *merging*
  - Allows optimizer to explore more plans, e.g, enabled access paths or consider more join orders

View Merging

- *Simple view*
  - Select-Project-Join
  - Merged automatically as it is always better.
- *Complex view*
  - Aggregation / group by, distinct, or outer-join
  - Complex view merging was heuristic-based;
  - It is cost-based in 10g
- In the following examples, in-line views are used to make it easy to see the view definition.
- All optimizations related to views apply to both inline views and user-defined views.

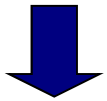
Select-Project-Join View Merging

```
SELECT t1.x, v.z
  FROM t1, t2, (SELECT t3.z, t4.m
                  FROM t3, t4
                 WHERE t3.k = t4.k AND t4.q = 5) v
 WHERE t2.p = t1.p AND t2.m = v.m;
```

```
SELECT t1.x, t3.z
  FROM t1, t2, t3, t4
 WHERE t2.p = t1.p AND t2.m = t4.m AND t3.k = t4.k AND t4.q = 5;
```

CVM - complex view merging

```
SELECT e1.last_name, e1.salary, v.avg_salary
   FROM employees e1,
   (SELECT department_id, avg(salary) avg_salary
      FROM employees e2
      GROUP BY department_id) v
WHERE e1.department_id = v.department_id
AND e1.salary > v.avg_salary;
```



```
SELECT e1.last_name last_name,
       e1.salary salary, avg(e2.salary) avg_salary
FROM hr.employees e1, hr.employees e2
WHERE e1.department_id = e2.department_id
GROUP BY e2.department_id,e1.rowid,e1.salary,e1.last_name
HAVING e1.salary > avg(e2.salary)
```

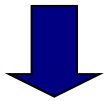
PM - predicate move-around

PM - predicate move-around (1)

- Moves inexpensive predicates into view query blocks in order to perform earlier filtering.
- Generates filter predicates based on
  - transitivity or
  - functional dependencies.
- Filter predicates are moved through SPJ, GROUP BY, DISTINCT views and views with OLAP constructs
- Copies of filter predicates can be moved up, down, and across query blocks.
- Enables new access paths and reduce the size of data that is processed later in more costly operations like joins or aggregations.
- It is performed automatically

PM - predicate move-around (2)

```
SELECT v1.k1, v2.q, max1
  FROM (SELECT   t1.k AS k1, MAX (t1.a) AS max1
           FROM t1, t2
          WHERE t1.k = 6 AND t1.z = t2.z
       GROUP BY t1.k) v1,
       (SELECT t1.k AS k2, t3.q AS q
          FROM t1, t3
         WHERE t1.y = t3.y AND t3.z > 4) v2
 WHERE v1.k1 = v2.k2 AND max1 > 50;
```



```
SELECT v1.x, v2.q, max1
  FROM (SELECT   t1.k AS k1, MAX (t1.a) AS max1
           FROM t1, t2
          WHERE t1.k = 6 AND t1.z = t2.z AND t1.a > 50
       GROUP BY t1.k) v1,
       (SELECT t1.k AS k2, t3.q AS q
          FROM t1, t3
         WHERE t1.y = t3.y AND t3.z > 4 AND t1.k = 6) v2
 WHERE v1.k1 = v2.k2;
```

JPPD - join predicate push-down

JPPD - join predicate push-down (1)

- Many types of views can not be merged; e.g., views containing UNION ALL/UNION; anti-/semi-joined views; some outer-joined views
- As an alternative, join predicates can be pushed inside unmerged views
- A pushed-down join predicate acts as a correlating condition inside the view and opens up new access paths e.g., index based nested-loop join
- Decision to do JPPD is cost-based

JPPD - join predicate push-down (2)

```
SELECT t1.c, t2.x
  FROM t1, t2, (SELECT t4.x, t3.y
                  FROM t4, t3
                 WHERE t3.p = t4.q AND t4.k > 4) v
 WHERE t1.c = t2.d AND t1.x = v.x(+) AND t2.d = v.y(+);
```

```
SELECT t1.c, t2.x
  FROM t1,
       t2,
    (SELECT t4.x, t3.y
       FROM t4, t3
      WHERE t3.p = t4.q AND t4.k > 4 AND t1.x = t4.x AND t2.d = t3.y) v
 WHERE t1.c = t2.d;
```

JF – Join Factorization

JF – Join Factorization

- Purpose:
  - Branches of UNION / UNION ALL that join a common table are combined to reduce the number of accesses to this common table.
- If this transformation is applied then the VW_JF* in the execution plan is a result of the join factorization.

JF – Join Factorization

```
(SELECT A1.C1 C1, A2.C2 C2
   FROM JOC.A1 A1, JOC.A2 A2
  WHERE A1.C1 = A2.C3 AND A1.C1 > 1)
UNION ALL
(SELECT A1.C1 C1, A2.C2 C2
   FROM JOC.A1 A1, JOC.A2 A2
  WHERE A1.C1 = A2.C3 AND A1.C1 > 20)
```

```
SELECT VW_JF_SEL$906F71F0.C1 C1, VW_JF_SEL$906F71F0.C2 C2
  FROM (SELECT VW_JF_SET$48F2D741.ITEM_2 C1, A2.C2 C2
          FROM ( (SELECT A1.C1 ITEM_1, A1.C1 ITEM_2
                    FROM JOC.A1 A1
                   WHERE A1.C1 > 1)
                 UNION ALL
                 (SELECT A1.C1 ITEM_1, A1.C1 ITEM_2
                    FROM JOC.A1 A1
                   WHERE A1.C1 > 20)) VW_JF_SET$48F2D741,
                 JOC.A2 A2
         WHERE VW_JF_SET$48F2D741.ITEM_1 = A2.C3) VW_JF_SEL$906F71F0
```

union all operation

JF – Join Factorization

```
(SELECT A1.C1 C1, A2.C2 C2
   FROM JOC.A1 A1, JOC.A2 A2
  WHERE A1.C1 = A2.C3 AND A1.C1 > 1)
UNION ALL
(SELECT A1.C1 C1, A2.C2 C2
   FROM JOC.A1 A1, JOC.A2 A2
  WHERE A1.C1 = A2.C3 AND A1.C1 > 20)
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                      |    10M|   300M|  7568  (11)| 00:00:22 |
|*  1 |  HASH JOIN           |                      |    10M|   300M|  7568  (11)| 00:00:22 |
|   2 |   VIEW               | VW_JF_SET$48F2D741   |     2 |    52 |  5008   (8)| 00:00:15 |
|   3 |    UNION-ALL         |                      |       |       |            |          |
|*  4 |     TABLE ACCESS FULL| A1                   |     1 |     2 |  2504   (8)| 00:00:08 |
|*  5 |     TABLE ACCESS FULL| A1                   |     1 |     2 |  2504   (8)| 00:00:08 |
|   6 |   TABLE ACCESS FULL  | A2                   |  5242K|    20M|  2377   (8)| 00:00:07 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("ITEM_1"="A2"."C3")
   4 - filter("A1"."C1">1)
   5 - filter("A1"."C1">20)
```

JE - Join Elimination

JE - Join Elimination (1)

- Eliminate unnecessary joins if there are constraints defined on join columns. If join has no impact on query results it can be eliminated.
  - e.departmens_id is foreign key and joined to primary key d.department_id
- Eliminate unnecessary outer joins – doesn't even require primary key – foreign key relationship to be defined.

```
SQL> select e.first_name, e.last_name, e.salary
     from employees e,
          departments d
     where e.department_id = d.department_id;
```

```
-------------------------------------+-------------------------------------+
| Id  | Operation          | Name     | Rows  | Bytes | Cost  | Time       |
-------------------------------------+-------------------------------------+
| 0   | SELECT STATEMENT   |          |       |       |     3 |            |
| 1   |  TABLE ACCESS FULL | EMPLOYEES|   106 |  2332 |     3 |  00:00:01 |
-------------------------------------+-------------------------------------+
Predicate Information:
----------------------
1 - filter("E"."DEPARTMENT_ID" IS NOT NULL)
```

JE - Join Elimination (3)

- **Purpose of join elimination**
  - Usually people don't write such "stupid" statements directly
  - Such situations are very common when a view is used which contains a join and only a subset of columns is used and therefor a join operation is really not required at all.

- **Known Limitations** (Source: Optimizer group blog)
  - Multi-column primary key-foreign key constraints are not supported.
  - Referring to the join key elsewhere in the query will prevent table elimination. For an inner join, the join keys on each side of the join are equivalent, but if the query contains other references to the join key from the table that could otherwise be eliminated, this prevents elimination. A workaround is to rewrite the query to refer to the join key from the other table.

SJC – Set Join Conversion

SJC - Set-Join Conversion

- Conversion of a set operator to a join operator.
- Disabled by default in 11gR2

- To enable it there are three options:
  - `alter session set "_convert_set_to_join"=true;`
  - `/*+ OPT_PARAM('_convert_set_to_join‘,'true') */`
  - `/*+ SET_TO_JOIN */`

No SJC By Default

```
select c4  from t1 minus select c2 from t2 ;
```

```
-----------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |  1000 |  6000 |     8  (63)| 00:00:01 |
|   1 |  MINUS              |      |       |       |            |          |
|   2 |   SORT UNIQUE       |      |  1000 |  3000 |     4  (25)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
|   4 |   SORT UNIQUE       |      |  1000 |  3000 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL| T2   |  1000 |  3000 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
```

SJC with OPT_PARAM hint

```
select /*+ opt_param('_convert_set_to_join','true') */ x.c4
from t1 x
minus
select y.c4
from t1 y;
```

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     3 |    18 |     8  (25)| 00:00:01 |
|   1 |  HASH UNIQUE        |      |     3 |    18 |     8  (25)| 00:00:01 |
|*  2 |   HASH JOIN ANTI    |      |    10 |    60 |     7  (15)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(SYS_OP_MAP_NONNULL("X"."C4")=SYS_OP_MAP_NONNULL("Y"."C4"))
```

SJC with SET_TO_JOIN Hint

```
select /*+ SET_TO_JOIN */ x.c4
from t1 x
minus
select y.c4
from t1 y;
```

```
-----------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     3 |    18 |     8  (25)| 00:00:01 |
|   1 |  HASH UNIQUE        |      |     3 |    18 |     8  (25)| 00:00:01 |
|*  2 |   HASH JOIN ANTI    |      |    10 |    60 |     7  (15)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(SYS_OP_MAP_NONNULL("X"."C4")=SYS_OP_MAP_NONNULL("Y"."C4"))
```

OBYE - Order BY Elimination

OBYE - order by elimination (1)

- OBYE operation eliminates unnecessary order by opertion from the SQL statement

```
select /*+ qb_name( main ) */ count(*) from (
   select /*+ qb_name( q1 ) */ p.prod_name
   from products p
   order by p.prod_name
);
```

```
-------------------------------------------------------------------------------
| Id  | Operation                          | Name                      | Rows  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                           |     1 |
|   1 |  SORT AGGREGATE                    |                           |     1 |
|   2 |   BITMAP CONVERSION COUNT          |                           |    72 |
|   3 |    BITMAP INDEX FAST FULL SCAN     | PRODUCTS_PROD_STATUS_BIX  |       |
-------------------------------------------------------------------------------
```

CNT - count(col) to count(*) transformation

CNT - count(col) to count(*) transformation

```
SQL> create table t1 (c1 number not null);
SQL> select count(c1) from t1;
```

```
CNT:   Considering count(col) to count(*) on query block
   SEL$1 (#0)
**********************
Count(col) to Count(*) (CNT)
**********************
CNT:     Converting COUNT(C1) to COUNT(*).
CNT:     COUNT() to COUNT(*) done.
```

- All rows should have a value and therefore Oracle can simply count the number of rows
- There is no need to actually retrieve the column value.

CNT - count(col) to count(*) transformation

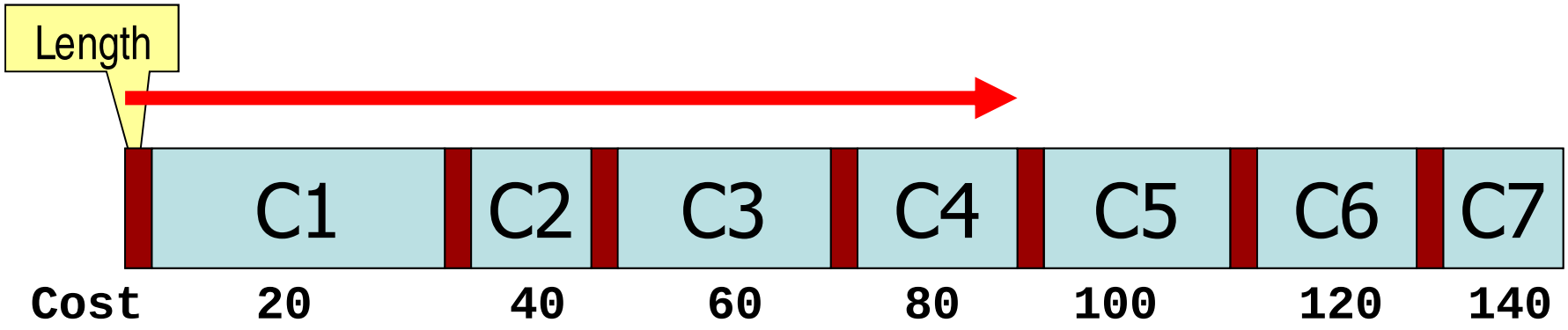
```
SQL> alter table t1 add (c2 varchar2(10)); /* nullable col */

SQL> select count(c2) from t1;
```

From CBO trace:

```
CNT:   Considering count(col) to count(*) on query block SEL$1 (#0)
**************************
Count(col) to Count(*) (CNT)
**************************
CNT:     COUNT() to COUNT(*) not done.
query block SEL$1 (#0) unchanged
```

CBO's Column Retrieval Cost

- Oracle stores columns in variable length format
- Each row is parsed in order to retrieve one or several columns.
- Each parsed column introduces cost of 20 CPU cycles regardless if it will be extracted or not.



CNT - count(col) to count(*) transformation

- Comparing the calculated cost from CBO trace file

  –Without CNT Transformation

```
Cost: 34.4695  Degree: 1  Card: 56229.0000  Bytes: 224916
Resc: 34.4695  Resc_io: 34.0000  Resc_cpu: 10399260
```

  –With CNT transformation the CPU cost is reduced

```
Cost: 34.4187  Degree: 1  Card: 56229.0000  Bytes: 0
Resc: 34.4187  Resc_io: 34.0000  Resc_cpu: 9274680
```

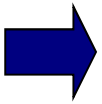
- The cost is reduced for 20 CPU cycles per row – Oracle has less work to do – accesses only the row directory and the row header in database block and doesn't need to parse the row data.

DP – Distinct Placement

GBP/DP – Group By/Distinct Placement

- Distinct placement
  - Purpose of this transformation is to eliminate duplicates as soon as possible.
- Group by placement
  - Eliminate
  - There is also option for group-by placement is a cost-based query transformation that is available as of 11.1.0.6

```
SELECT t1.n, t2.n, count(*)
FROM t1, t2
WHERE t1.id = t2.t1_id
GROUP BY t1.n, t2.n
```



```
SELECT t1.n, vw_gb.n, sum(vw_gb.cnt)
FROM t1, (SELECT t2.t1_id, t2.n, count(*)
AS cnt
          FROM t2
          GROUP BY t2.t1_id, t2.n) vw_gb
WHERE t1.id = vw_gb.t1_id
GROUP BY t1.n, vw_gb.n
```

GBP/DP – Distinct Placement (1)

```
select distinct t3.c1, t4.c1
from t3, t4
where t3.id = t4.id;
```

```
SELECT DISTINCT "T3"."C1" "C1",
       "VW_DTP_B2E7E6CA"."ITEM_2" "C1"
FROM
       (SELECT DISTINCT "T4"."ID" "ITEM_1",
              "T4"."C1" "ITEM_2"
        FROM "JOC"."T4" "T4") "VW_DTP_B2E7E6CA",
             "JOC"."T3" "T3"
       WHERE "T3"."ID"="VW_DTP_B2E7E6CA"."ITEM_1"
```

GBP/DP – Distinct Placement (2)

```
-----------------------------------------------------+-----------------------------------+
| Id  | Operation                      | Name        | Rows  | Bytes | Cost  | Time      |
-----------------------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT               |             |       |       |  8237 |           |
| 1   |  HASH UNIQUE                   |             |     1 |   410 |  8237 |  00:02:39 |
| 2   |   NESTED LOOPS                 |             |       |       |       |           |
| 3   |    NESTED LOOPS                |             |    72 |   29K |  8236 |  00:02:39 |
| 4   |     VIEW                       | VW_DTP_B2E7E6CA|    72 |   14K |  8091 |  00:02:38 |
| 5   |      HASH UNIQUE               |             |    72 |   14K |  8091 |  00:02:38 |
| 6   |       TABLE ACCESS FULL        | T4          |  977K |  195M |  8046 |  00:02:37 |
| 7   |     INDEX RANGE SCAN           | T3_I        |     1 |       |     2 |  00:00:01 |
| 8   |    TABLE ACCESS BY INDEX ROWID | T3          |     1 |   206 |     3 |  00:00:01 |
-----------------------------------------------------+-----------------------------------+
Predicate Information:
----------------------
7 - access("T3"."ID"="ITEM_1")
```

- Related optimizer hints:
  - PLACE_DISTINCT
  - NO_PLACE DISTINCT

SLP – Select List Pruning

Select List Prunning

- The purpose of select list pruning is to remove unnecessary columns or expressions from the SELECT clause of subqueries, inline view, or regular views.

```
select CUST_ID, CUST_FIRST_NAME, CUST_LAST_NAME
from
   ( select *
     from sh.customers
     where cust_id in (49671,3228)
   );
```

SLP: Select List Pruning

```
select count(*) from (select c1, max(id) max_id from t3);
```

Not a valid statement – missing group by expression

```
SLP: Removed select list item C1 from query block SEL$2
query block SEL$1 (#0) unchanged

Final query after transformations:
******* UNPARSED QUERY IS *******
SELECT COUNT(*) "COUNT(*)"
FROM
    (SELECT MAX("T3"."ID") "MAX_ID"
    FROM "JOC"."T3" "T3") "from$_subquery$_001"
```

By pruning select list turned into valid SQL statement

Oracle 12c Query Transformations

Scalar Subquery Unnesting transformation

```
SELECT
    u.username,
    (SELECT MAX(created) FROM test_objects o WHERE o.owner = u.username)
FROM
    test_users u
WHERE
    username LIKE 'JOC%';
```

Scalar Subquery Unnesting transformation

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |              |       |       |   536 (100)|          |
|   1 |  HASH GROUP BY      |              |     1 |    35 |   536   (1)| 00:00:01 |
|*  2 |   HASH JOIN OUTER   |              |    75 |  2625 |   536   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL| TEST_USERS   |     1 |    21 |     3   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| TEST_OBJECTS |  2933 | 41062 |   533   (1)| 00:00:01 |
------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("O"."OWNER"="USERNAME")
   3 - filter("USERNAME" LIKE 'JOC%')
   4 - filter("O"."OWNER" LIKE 'JOC%')
```

Null Accepting Semi-Joins

```
SQL> SELECT * FROM t1
  2  WHERE EXISTS (SELECT 1 FROM t2 WHERE t1.a=t2.b)
  3  OR t1.a IS NULL;

         A          C
---------- ----------
         1          1
                    2


Final query after transformations:******* UNPARSED QUERY IS *******
SELECT "T1"."A" "A","T1"."C" "C"
FROM "JOC"."T2" "T2","JOC"."T1" "T1"
WHERE "T1"."A"="T2"."B"
```

Null Accepting Semi-Joins

```
-------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |       |       |     6 (100)|          |
|*  1 |  HASH JOIN SEMI NA  |      |     1 |    39 |     6   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| T1   |     2 |    52 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T2   |     2 |    26 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("T1"."A"="T2"."B")

Note

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------
- dynamic statistics used: dynamic sampling (level=2)
```

Null Accepting Semi-Joins

- From CBO Trace

```
HA Join
  HA cost: 6.02
     resc: 6.02 resc_io: 6.00 resc_cpu: 672354
     resp: 6.02 resp_io: 6.00 resp_cpu: 672354
Best:: JoinMethod: HashNullAcceptingSemi
       Cost: 6.02  Degree: 1  Resp: 6.02  Card: 1.00
Bytes: 39
```

- Hidden parameter:
  - _optimizer_null_accepting_semijoin = true

Conclusions

Conclusions

1. Help CBO by defining all possible constraints. CBO uses them extensively during the SQL statement transformations. Telling more “truth” to CBO usually helps.

2. Feed the CBO with accurate statistics, only for complex expressions use dynamic sampling.

3. Misestimated cardinality in Cost Based Transformation leads to sub-optimal plan.

4. Use transformation techniques when rewriting the statement to obtain optimal plan. One can even use **NO_QUERY_TRANSFORMATION** hint to disable all transformations during optimization.

Thank you for your interest!

Q&A